Práctica 2: Búsquedas con trayectorias múltiples
Selección de características

Primero el mejor, enfriamiento simulado y búsqueda tabú básica

Alejandro García Montoro
76628233F, agarciamontoro@correo.ugr.es

Grupo de los viernes a las 17.30

Curso 2015 - 2016

Índice

1. Descripción del problema

La selección de características es una técnica muy usada en problemas de aprendizaje automático.

El aprendizaje automático, visto de una forma muy general, tiene como objetivo clasificar un conjunto de objetos —modelador por una serie de atributos— en clases.

Esta clasificación se aprende desde los datos, pero la selección de los atributos que definen la modelización del objeto puede no ser la más apropiada: en ocasiones hay atributos superfluos o demasiado ruidosos que sería conveniente eliminar. Además, cuantos menos atributos definan un objeto, más rápido y preciso será el aprendizaje. Es aquí entonces donde aparece la pregunta que guia todo este trabajo: ¿cómo identificar los atributos que mejor aprendizaje promueven?

La respuesta a esta pregunta pasa por la selección de características, cuyo objetivo es reducir la definición de un objeto a una serie de características que faciliten el aprendizaje.

La idea es entonces la siguiente: dado un conjunto de m objetos definidos por un conjunto C de n características, y considerado un modelo de aprendizaje f que intenta aprender la clasificación de estos objetos, encontrar el subconjunto $C' \subset C$ que maximiza el modelo f.

Así, vemos claramente que el tamaño de caso de nuestro problema es n —el número de características— y que el objetivo está bien definido: eliminar aquellas características que o bien empeoren la bondad de f o bien sean innecesarias.

Con todos estos elementos definidos, podemos pasar a analizar las metaheurísticas consideradas.

2. Metaheurísticas

2.1. Introducción

Los algoritmos considerados para resolver el problema son los siguientes:

- Búsqueda multiarranque básica (BMB).
- *Greedy randomized adaptive search procedure (GRASP).*
- Búsqueda local reiterada (*ILS*).

Además, compararemos estas metaheurísticas con el algoritmo voraz *Sequential forward selection.*

Estas tres metaheurísticas reúnen las condiciones necesarias para resolver el problema: el espacio de soluciones de nuestro problema puede ser analizado mediante las estructuras de generación de vecinos y los criterios de aceptación que utilizan estos algoritmos. Veamos con un poco más de detalle los aspectos comunes a las metaheurísticas implementadas:

Datos de entrada

Todos los algoritmos considerados reciben un conjunto de entrenamiento cuyos objetos tienen la siguiente estructura:

$$(s_1, s_2, \ldots, s_n, c)$$

donde $(s_1, s_2, \ldots, s_n)$ es el conjunto de valores de los atributos que definen el objeto y c la clase a la que pertenece.

Esquema de representación

El espacio de soluciones S de nuestro problema es el conjunto de todos los vectores s de longitud n —el número de características— binarios; es decir:

$$S = \{s = (s_1, s_2, \ldots, s_n)/s_i \in \{0, 1\}\ \forall i = 1, 2, \ldots, n\}$$

La posición i-ésima de un vector $s \in S$ indicará la inclusión o no de la característica i-ésima en el conjunto final C'.

Función objetivo

La finalidad de las metaheurísticas será maximizar la función objetivo siguiente:

$$\begin{aligned} f\colon S &\to [0, 100] \\ s &\mapsto f(s) = \text{Acierto del 3-NN sobre s} \end{aligned}$$

$f(s)$ es, por tanto, la tasa de acierto del clasificador 3-NN producido a partir de la solución s.

El clasificador 3-NN es una particularización del clasificador k-NN, que mide la distancia de la instancia considerada a todos los demás objetos en el conjunto de datos de entrenamiento y le asigna la clasificación mayoritaria de entre los k vecinos más cercanos; esto es:

Pseudocódigo 1 Clasificador k-NN

```
1: function k-NN(instance, trainingData)
2:     distances ← euclideanDistance(instance, trainingData)
3:     neighbours ← getClosestNeighbours(distances)
4:     classification ← mostVotedClassification(neighbours)
5:     return classification
```

Así, dada una solución $s \in S$, la función objetivo es como sigue:

Pseudocódigo 2 Función objetivo

```
1: function f(s, train, target)
2:     samples ← removeZeroColumns(s, train)
3:     sum ← 0
4:     for instance ∈ samples do
5:         class ← k-NN(instance, samples)
6:         sum ← sum + { 1  if class = actualClass(instance, target)
                       { 0  if class ≠ actualClass(instance, target)
7:     return sum / (number of samples in train)
```

donde *removeZeroColumns(s, train)* elimina la columna i-ésima de *train* si y sólo si $s_i = 0$ y *actualClass(instance, target)* devuelve la clase real —no la aprendida— del objeto *instance*.

Entorno de soluciones

Dada una solución $s \in S$, el entorno de soluciones vecinas a s es el conjunto

$$E(s) = \{s' \in S / |s' - s| = (0, \ldots, 0, \underbrace{1}_{i}, 0, \ldots, 0), i \in \{1, 2, \ldots, n\}\}$$

es decir, $E(s)$ son las soluciones que difieren de s en una única posición. Es evidente entonces que el conjunto $E(S)$ tiene siempre exactamente cardinal igual a n.

El operador de generación de vecino de la solución s es entonces como sigue:

Pseudocódigo 3 Operador de generación de vecino

1: **function** FLIP(solution, feature)
2: $s' \leftarrow solution$
3: $s'[feature] \leftarrow (s'[feature] + 1) \bmod 2$
4: **return** s'

Criterios de parada

Aunque los criterios de parada dependerán de la metaheurística considerada —en general se parará cuando no se encuentre mejora en el entorno—, en todos los algoritmos pararemos necesariamente tras llegar a las 15000 evaluaciones con el clasificador 3-NN sobre las soluciones generadas.

Generación de soluciones aleatorias

En los algoritmos de búsqueda multiarranque básica y búsqueda local reiterada se genera una serie de soluciones aleatorias sobre las que se aplica búsqueda local de una u otra forma. La generación de estas soluciones aleatorias sigue el siguiente esquema:

Pseudocódigo 4 Generación de soluciones aleatorias

1: **function** RANDOMSOLUTION(size)
2: **for** $i \in 1, 2, \ldots, size$ **do**
3: random $\leftarrow$ uniformRandomNumber([0,1])
4: $s_i \leftarrow \begin{cases} 0 & \textbf{if } random \leq 0{,}5 \\ 1 & \textbf{if } random > 0{,}5 \end{cases}$
5: solution $\leftarrow (s_1, s_2, \ldots, s_{size})$
6: **return** solution

Búsqueda local

El algoritmo de búsqueda local considerado es el implementado para la primera práctica: la búsqueda local primero el mejor.

El método de exploración del entorno es el siguiente: dada una solución s, escogemos una característica al azar, aplicamos el operador $flip$ para obtener una solución vecina y medimos su bondad con $f(s)$; si es mejor que s, nos quedamos con ella como mejor solución y volvemos a empezar; si no, tomamos otra característica al azar —sin repetir— y seguimos el proceso.

Pararemos el algoritmo si: o bien al haber explorado el entorno completo de la solución actual ninguna de las soluciones vecinas es mejor, o bien si se han alcanzado 15000 iteraciones. Estaremos entonces ante un máximo —probablemente local— y el algoritmo no podrá seguir mejorando la solución.

El pseudocódigo de todo el procedimiento es el siguiente, donde hemos puesto un tercer argumento optativo, *initSol*, que por defecto es vacío, y en cuyo caso se genera una solución aleatoria como solución inicial. Si no es vacío, tomamos como solución inicial la *initSol*, de manera que la búsqueda se centrará en el entorno de esta solución.

Pseudocódigo 5 Búsqueda local primero el mejor

```
function bestFirst(train, target, initSol = ∅)
    if initSol = ∅ then
        s ← genInitSolution()
    else
        s ← initSol
    bestScore ← f(s, train, target)
    improvementFound ← True
    while improvementFound and iterations <15000 do
        improvementFound ← False
        for feature ← genRandomFeature(s) do    ▷ Without replacement
            s'← genNeighbour(s, feature)
            score ← f(s', train, target)
            if score > bestScore then
                s, bestScore ← s', score
                improvementFound ← True
                break for
    return s, bestScore
```

2.2. Búsqueda multiarranque básica

La búsqueda multiarranque básica es el primer algoritmo considerado, cuyo comportamiento es muy sencillo: se trata de generar un número N de soluciones aleatorias y, para cada una de ellas, explotar su entorno de soluciones con el algoritmo de búsqueda local primero el mejor.

La idea que intenta perseguir este algoritmo es clara: como la búsqueda local ya aporta la suficiente intensificación en zonas locales del espacio de búsqueda, se intenta aumentar la diversidad para explorar zonas diferentes. Esto último se consigue con la generación aleatoria de las N soluciones, que previsiblemente cubren una zona mayor del espacio.

Durante todo el proceso mantendremos la mejor solución encontrada hasta el momento, de manera que al terminar el algoritmo —tras buscar localmente desde 25 soluciones aleatorias y con un máximo de 15000 iteraciones en cada búsqueda local— se devuelve la mejor encontrada.

Como el procedimiento *bestFirst* genera una solución aleatoria inicial si no se le pasa un tercer argumento, todo el cómputo del algoritmo BMB se encuentra recogido en esa llamada. Sólo tenemos que preocuparnos de repetir

el procedimiento N veces, donde N es, en este caso, igual a 25.

El procedimiento se puede ver en el Pseudocódigo 6.

Pseudocódigo 6 Búsqueda multiarranque básica

```
1: function BMB(train, target)
2:     bestSolution, bestScore ← ∅, -1
3:     for i ∈ {1, 2, ..., 25} do
4:         currentSolution, currentScore ← bestFirst(train, target)
5:         if currentScore >bestScore then
6:             bestSolution, bestScore ← currentSolution, currentScore
7:     return bestSolution, bestScore
=0
```

2.3. *Greedy randomized adaptive search procedure*

La estructura del algoritmo GRASP es similar a la de la búsqueda multiarranque básica: tras generar una solución inicial, se ejecuta búsqueda local sobre ella para mejorarla. La diferencia con el algoritmo anterior se encuentra, precisamente, en cómo se genera la solución inicial: si bien en BMB se hacía de forma aleatoria, aquí se añade un paso intermedio: tras generar una solución aleatoria, se ejecuta una variante del algoritmo *SFS* que en cada iteración elige, de entre las características que más ganancia aportan, una de forma aleatoria.

Antes de entrar en detalles, es conveniente ver el procedimiento general en el Pseudocódigo 7.

Pseudocódigo 7 GRASP

```
1: function BMB(train, target)
2:     bestSolution, bestScore ← ∅, -1
3:     for i ∈ {1, 2, ..., 25} do
4:         currentSolution, currentScore ← randomSFS(train, target)
5:         currentSolution, currentScore ← bestFirst(train, target, curren-
   tSolution)
6:         if currentScore >bestScore then
7:             bestSolution, bestScore ← currentSolution, currentScore
8:     return bestSolution, bestScore
```

Como vemos, sólo se ha añadido la generación de la solución inicial antes de la búsqueda local. En la llamada a *randomSFS* se encuentra el núcleo de este algoritmo, así que echémosle un vistazo más de cerca.

Algoritmo voraz probabilístico

La generación de la solución inicial en el algoritmo *GRASP* se hace con un algoritmo voraz probabilístico, cuya idea central es la siguiente: en cada iteración se evalúan todas las características aún no seleccionadas, almacenando para cada una de ellas la ganancia —que puede ser negativa— que produce con respecto a la solución actual; tras esta evaluación, se genera una lista restringida de candidatos definida a partir de la peor y mejor ganancia registradas; de entre esta lista, se toma una característica al azar y se añade a la solución, terminando así la iteración.

La generación de la lista restringida de candidatos, LRC, se hace en base al siguiente umbral:

$$\mu = \max_i\{g_i\} - \alpha(\max_i\{g_i\} - \min_i\{g_i\})$$

donde g_i es la ganancia que produce añadir la característica i-ésima a la solución actual y, en este caso, se ha tomado $\alpha = 0{,}3$.

Podemos ya ver el procedimiento del algoritmo voraz probabilístico en el Pseudocódigo 8.

Pseudocódigo 8 Algoritmo voraz probabilístico

```
1:  function RANDOMSFS(train, target)
2:      s ← genZeroSolution()
3:      bestScore ← 0
4:      while there was improvement with some feature do
5:          g ← (0, 0, ..., 0)          ▷ Size = number of not selected features
6:          for every feature f in not selected features do
7:              s ← addFeature(s,f)
8:              currentScore ← f(s, train, target)
9:              gain ← currentScore - bestScore
10:             g_f ← gain
11:             s ← removeFeature(s,f)
12:         μ ← máx_i{g_i} − α(máx_i{g_i} − mín_i{g_i})
13:         LRC ← {f ∈ not selected features / g_f > μ}
14:         f ← random choice from LRC
15:         if g_f > 0 then
16:             s ← addFeature(s,f)
17:             bestScore ← bestScore + bestGain
18:     return s, bestScore
```

2.4. Búsqueda local reiterada

El algoritmo de búsqueda local reiterada, o ILS por sus siglas en inglés, busca también introducir algo de diversidad en las soluciones exploradas.

Para ello, parte de una solución inicial aleatoria que mejora con búsqueda local. A partir de ahí se sigue un procedimiento iterativo repetido N veces —en nuestro caso, tomaremos $N = 24$, ya que queremos llamar 25 veces a la búsqueda local—:

- Se muta la solución previa.
- Se realiza búsqueda local con esa mutación como solución inicial.
- Se actualiza la mejor solución.

Podemos ver este procedimiento con más detalle en el Pseudocódigo 9.

Pseudocódigo 9 Búsqueda local reiterada

```
1: function ILS(train, target)
2:     bestSolution, bestScore ← bestFirst(trian, target)
3:     prevSolution, prevScore ← bestSolution, bestScore
4:     for i ∈ {1, 2, ..., 24} do
5:         mutation ← mutateSolution(prevSolution)
6:         currentSolution, currentScore ← bestFirst(train, target, currentSolution)
7:         if currentScore >prevScore then
8:             prevSolution, prevScore ← currentSolution, currentScore
9:         if prevScore >bestScore then
10:            bestSolution, bestScore ← prevSolution, prevScore
11:    return bestSolution, bestScore
```

Mutación

La mutación de las soluciones llevada a cabo en el algoritmo ILS es simple: basta tomar un 10 % de características de forma aleatoria y cambiar su estado: si están a 1 ponerlas a 0 y viceversa; esto es, aplicar el operador *flip* para cada una de ellas.

Este sencillo procedimiento puede verse en el Pseudocódigo 10.

Pseudocódigo 10 Mutación para la ILS

```
1: function MUTATESOLUTION(solution, p = 0.1)
2:     n ← size of solution
3:     indices ← Take ⌈pn⌉ random indices in {1, ..., n}
4:     for i ∈ indices do
5:         flip(solution, i)
6:     return solution
```

2.5. Algoritmo de comparación

Para la comparación de los algoritmos implementados consideraremos el algoritmo voraz *Sequential forward selection*, que se puede ver en el Pseudocódigo 11.

Pseudocódigo 11 Algoritmo de comparación

```
1:  function sequentialForwardSelection(train, target)
2:      s ← genZeroSolution()
3:      bestScore ← 0
4:      while there was improvement with some feature do
5:          for every feature f in not selected features do
6:              s ← addFeature(s,f)
7:              currentScore ← score(s, train, target)
8:              if currentScore > bestScore then
9:                  bestScore ← currentScore
10:                 bestFeature ← f
11:             s ← removeFeature(s,f)
12:         if there was a best feature f then
13:             s ← addFeature(s,f)
14:     return s, bestScore
```

La idea es la siguiente: en cada iteración escogemos la característica, de entre las aún no seleccionadas, que mejor valor de la función objetivo produce, si y sólo si este valor es mejor que el actual.

3. Desarrollo de la práctica

La práctica se ha desarrollado por completo en Python, definiendo cada algoritmo en una función diferente con cabeceras iguales —mismo número y tipo de parámetros— para poder automatizar el proceso de recogida de datos.

3.1. *Framework* de aprendizaje automático

Se ha usado, además, el módulo *Scikit-learn*, del que se ha usado la siguiente funcionalidad:

- Particionamiento de los datos. *Scikit-learn* aporta una función para hacer un particionado aleatorio de los datos en una parte de aprendizaje y otra de test. Esto se ha usado para implementar la técnica 5×2 *cross-validation*.

3.2. Paralelización en GPU de la función objetivo

Aunque en la práctica anterior se usó también *Scikit-learn* para medir la función objetivo, la lentitud de este proceso me llevó a buscar otras alternativas: después de intentar usar el mismo módulo con la opción de paralelización CPU y conseguir prácticamente los mismos resultados —para notar mejoría, dicen los desarrolladores, haría falta trabajar con bases de datos con varios miles de muestras—, decidí buscar una solución propia.

Como gracias a mi Trabajo fin de grado he aprendido a hacer computación general paralelizada en GPU, decidí usar la librería CUDA —y en concreto su interfaz para Python, PyCUDA— para implementar la función objetivo de una forma eficiente. La mejoría en tiempo conseguida es muy notable —es del orden de 20 a 100 veces más rápido[1]— y, tras muchas pruebas para comprobar que el cálculo de la función era correcto, sustituí el k-NN de *Scikit-learn* con el implementado en CUDA.

Todo este trabajo, necesario para el correcto funcionamiento de la práctica, se encuentra en los ficheros bajo el directorio *src/knnGPU*, que contienen la implementación en C del k-NN y la interfaz para poder usar el código desde Python.

Además, como vi que este código podía beneficiar a mis compañeros, decidí publicarlo de forma abierta en un repositorio de Github[2], bien documentado y con una guía de uso.

Gracias a esto, algunos amigos me ayudaron a mejorar el código: yo había implementado sólo la función objetivo sobre los datos de training, y Jacinto

[1] Los tiempos son muy dependientes del número de muestras de la base de datos y del número de características. Para tener una idea de la mejora, se pueden comparar los tiempos de las tablas 3-NN y SFS de esta y la anterior práctica.

[2] `https://github.com/agarciamontoro/metaheuristics`

Carrasco Castillo la modificó para poder hacer la medición también con los datos de test. Además, Luís Suárez Lloréns me ayudó a probar cambios que creíamos que iban a mejorar aún más la eficiencia —aunque tras mucho trabajo vimos que la implementación inicial era la más rápida—. Por último, Antonio Álvarez Caballero, Anabel Gómez Ríos y Gustavo Rivas Gervilla me ayudaron a testear el código, probándolo con sus algoritmos y los datos que tenían de anteriores prácticas.

3.3. Manual de usuario

Para la ejecución de la práctica es necesario tener instalado Python 3, el módulo *Scikit-learn*, *PyCUDA* y *jinja2* —estos dos últimos módulos son necesarios para la implementación del código paralelizado—, así como disponer de una tarjeta gráfica compatible con CUDA.

Todo se encuentra automatizado en el fichero *src/02_multiPath.py*, así que sólo es necesario ejecutar la siguiente orden desde el directorio raíz de la práctica: *python src/02_multiPath.py*

Así se ejecutarán todos los algoritmos con todas las bases de datos usando la ténica del 5×2 *cross-validation*. Las tablas generadas se guardarán en el directorio *results/02*.

La semilla utilizada se inicializa al principio de la ejecución del programa con las líneas *np.random.seed(19921201)* y *random.seed(19921201)*.

4. Análisis de resultados

En esta sección vamos a presentar los datos recogidos de la ejecución de todos los algoritmos con las tres bases de datos consideradas: *WDBC*, *Movement Libras* y *Arrhytmia*. Las bases de datos se han considerado completas en todos los casos, tal y como se nos entregaron —arreglando alguna columna defectuosa y homogeneizando el nombre de la columna de clasificación para poder automatizar el proceso—.

Para el análisis de cada algoritmo con cada base de datos se han generado cinco particiones aleatorias de los datos y se ha ejecutado el algoritmo considerando cada partición como datos de entrenamiento y test, con la técnica 5×2 *cross-validation*.

En cada una de estas ejecuciones se han medido los siguientes datos:

- Tasa de clasificación en la partición de entrenamiento —en %—.
- Tasa de clasificación en la partición de test —en %—.
- Tasa de reducción de las características —en %—.
- Tiempo de ejecución —en segundos—.

Veamos ya los datos y analicemos los resultados obtenidos:

4.1. Clasificador k-NN

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,1831	96,4912	0	$4{,}8 \cdot 10^{-3}$	67,2222	77,2222	0	$4{,}7 \cdot 10^{-3}$	64,5833	63,4021	0	0,02
Partición 1-2	96,4912	96,1268	0	$3{,}2 \cdot 10^{-3}$	70,5556	71,1111	0	$4{,}6 \cdot 10^{-3}$	65,4639	62,5	0	0,0182
Partición 2-1	94,3662	97,193	0	$3{,}2 \cdot 10^{-3}$	68,8889	75	0	$4{,}7 \cdot 10^{-3}$	62,5	63,4021	0	0,0196
Partición 2-2	97,193	95,7746	0	$3{,}3 \cdot 10^{-3}$	69,4444	80,5556	0	$4{,}6 \cdot 10^{-3}$	58,2474	63,0208	0	0,0181
Partición 3-1	95,7746	96,4912	0	$3{,}2 \cdot 10^{-3}$	67,2222	68,8889	0	$4{,}6 \cdot 10^{-3}$	66,6667	59,7938	0	0,0195
Partición 3-2	95,4386	96,1268	0	$3{,}4 \cdot 10^{-3}$	74,4444	71,1111	0	$4{,}7 \cdot 10^{-3}$	61,3402	63,0208	0	0,0181
Partición 4-1	96,831	95,4386	0	$3{,}2 \cdot 10^{-3}$	64,4444	72,2222	0	$4{,}6 \cdot 10^{-3}$	60,9375	62,8866	0	0,0196
Partición 4-2	96,4912	95,4225	0	$3{,}2 \cdot 10^{-3}$	67,7778	75,5556	0	$4{,}6 \cdot 10^{-3}$	65,4639	64,0625	0	0,0181
Partición 5-1	95,0704	96,8421	0	$3{,}2 \cdot 10^{-3}$	68,3333	71,1111	0	$4{,}7 \cdot 10^{-3}$	65,625	60,3093	0	0,0195
Partición 5-2	97,8947	95,7746	0	$3{,}2 \cdot 10^{-3}$	70	79,4444	0	$4{,}6 \cdot 10^{-3}$	63,4021	65,1042	0	0,0182
Medias	96,2734	96,1681	0	$3{,}4 \cdot 10^{-3}$	68,8333	74,2222	0	$4{,}7 \cdot 10^{-3}$	63,423	62,7502	0	0,0189

Cuadro 1: Datos del clasificador k-NN

En la tabla 1 se pueden ver los datos obtenidos del clasificador k-NN. La selección de características en este algoritmo es nula, ya que es la propia función objetivo considerando la totalidad de las características. Aún así, se ha añadido aquí para conocer la tasa de clasificación en los conjuntos de entrenamiento y de test considerando como solución la trivial: esto es, todas las características.

Como vemos, aunque en la primera base de datos las tasas de clasificación son buenas, en las otras dos son muy mejorables, lo que nos da una idea de la necesidad de la reducción de características.

4.2. Algoritmo de comparación

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,8873	95,4386	83,3333	0,2466	68,8889	71,1111	92,2222	0,7259	76,5625	66,4948	97,482	2,3164
Partición 1-2	97,5439	93,3099	80	0,2503	78,3333	68,8889	90	0,9295	84,0206	71,875	97,1223	2,6907
Partición 2-1	97,8873	95,7895	80	0,2503	80,5556	71,1111	88,8889	1,0394	78,125	61,3402	96,0432	3,8524
Partición 2-2	97,8947	93,3099	80	0,2507	76,6667	73,8889	88,8889	1,0389	82,9897	70,8333	96,7626	3,0664
Partición 3-1	97,5352	95,7895	80	0,25	75	71,6667	90	0,9323	76,5625	61,8557	96,7626	3,0453
Partición 3-2	97,193	90,8451	86,6667	0,1782	81,1111	72,7778	90	0,932	71,6495	71,3542	98,9209	1,0239
Partición 4-1	97,5352	95,7895	86,6667	0,1734	76,1111	71,6667	93,3333	0,6216	73,4375	72,1649	98,5612	1,3109
Partición 4-2	95,4386	92,6056	93,3333	0,0971	72,2222	65,5556	92,2222	0,7201	74,7423	70,8333	98,9209	1,0285
Partición 5-1	95,7746	96,8421	86,6667	0,1731	80	72,7778	87,7778	1,1445	84,8958	71,134	96,4029	3,395
Partición 5-2	97,8947	93,3099	86,6667	0,1737	73,8889	67,7778	87,7778	1,1493	82,9897	75	96,4029	3,4243
Medias	97,2585	94,3029	84,3333	0,2043	76,2778	70,7222	90,1111	0,9234	78,5975	69,2886	97,3381	2,5154

Cuadro 2: Datos del algoritmo *Sequential forward selection*

En la tabla 2 vemos los resultados del algoritmo de comparación, el *Sequential forward selection*. Este algoritmo voraz tiene una alta tasa de reducción de características, pero la tasa de clasificación no mejora la del clasificador con la solución trivial.

Esto se debe a que consideramos cada característica de una forma secuencial, y una vez seleccionamos una, es imposible descartarla. Aún así, este algoritmo podría ser interesante si lo que buscamos es una reducción drástica del número de características —como vemos, sobre el 80 %— sin perder mucha información —las tasas de clasificación son más o menos iguales a las del clasificador con la solución trivial—.

4.3. BMB

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	99,2958	95,4386	40	3,8023	74,4444	76,6667	53,3333	12,1475	74,4792	65,9794	53,9568	128,3378
Partición 1-2	98,2456	95,0704	36,6667	2,9134	79,4444	72,2222	52,2222	12,3056	72,1649	64,0625	53,9568	116,3462
Partición 2-1	97,8873	96,8421	56,6667	3,3573	77,7778	76,1111	47,7778	11,0225	73,9583	67,0103	45,6835	136,2465
Partición 2-2	98,5965	95,0704	50	3,1442	78,3333	76,1111	50	12,4279	72,6804	60,4167	56,1151	148,9987
Partición 3-1	97,8873	95,7895	50	3,056	76,6667	70	46,6667	13,6095	76,0417	64,433	46,7626	156,8745
Partición 3-2	97,8947	95,4225	56,6667	3,2644	80	73,8889	54,4444	11,5957	70,6186	60,9375	52,518	124,0632
Partición 4-1	98,9437	95,7895	46,6667	3,2637	76,6667	75	52,2222	11,5345	71,875	62,8866	53,9568	140,4098
Partición 4-2	97,8947	97,5352	46,6667	3,6167	76,6667	76,6667	54,4444	12,7171	73,7113	64,5833	50,3597	110,7842
Partición 5-1	97,8873	97,8947	36,6667	3,2381	75,5556	72,2222	58,8889	12,1538	75	61,3402	43,1655	125,6062
Partición 5-2	98,9474	94,7183	53,3333	2,9365	75,5556	76,6667	51,1111	12,6701	72,6804	65,625	51,0791	121,7811
Medias	98,348	95,9571	47,3333	3,2593	77,1111	74,5556	52,1111	12,2184	73,321	63,7274	50,7554	130,9448

Cuadro 3: Datos de la búsqueda multiarranque básica

En la tabla 3 vemos los datos de la primera metaheurística real considerada: la búsqueda multiarranque básica.

Esta metaheurística consigue unas tasas de clasificación algo mejores —excepto en la base de datos Arrythmia, cuyo rendimiento se queda a niveles de la solución trivial con todas las características—-.

Los tiempos, evidentemente, son mucho mayores que en el algoritmo de comparación: la búsqueda local es computacionalmente intensa, y se repite 25 veces sobre zonas diferentes del espacio de búsqueda.

Aunque las tasas de reducción son bastante aceptables —un 50 % de características son eliminadas— la clasificación fuera de la muestra de entrenamiento puede mejorarse aún mucho más. Esto se puede deber al hecho de que la búsqueda local no puede sali del entorno de cada una de las soluciones iniciales.

El intento de este algoritmo de añadir diversidad es bueno, pero quizá algo insuficiente: no se consigue una gran mejora.

4.4. Greedy randomized adaptive search procedure

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	99,2958	95,4386	73,3333	7,0661	76,1111	75	83,3333	28,8141	82,2917	74,2268	94,6043	40,6884
Partición 1-2	98,9474	95,0704	76,6667	6,935	81,1111	75	82,2222	28,9204	84,0206	64,5833	93,1655	65,2212
Partición 2-1	97,8873	97,8947	70	7,4439	80	74,4444	84,4444	29,7892	81,7708	68,0412	94,964	44,9096
Partición 2-2	99,2982	92,6056	76,6667	6,8884	82,7778	75	83,3333	30,6671	84,5361	75,5208	92,8058	62,7484
Partición 3-1	98,2394	95,7895	76,6667	7,3749	78,8889	68,8889	85,5556	28,8052	82,2917	69,0722	93,8849	67,0018
Partición 3-2	98,2456	92,2535	80	5,8633	83,3333	71,6667	85,5556	29,7811	83,5052	70,3125	93,1655	44,4038
Partición 4-1	98,9437	95,0877	76,6667	7,0251	81,6667	73,3333	83,3333	27,3301	82,8125	72,6804	94,6043	41,221
Partición 4-2	97,8947	94,3662	76,6667	6,3724	80	73,3333	84,4444	28,6023	84,5361	72,3958	94,6043	61,2677
Partición 5-1	98,2394	97,193	76,6667	8,1625	78,3333	70,5556	86,6667	29,9844	85,4167	69,5876	93,8849	60,0604
Partición 5-2	98,9474	96,1268	70	6,5719	78,8889	69,4444	78,8889	31,7094	81,9588	77,6042	93,5252	45,1128
Medias	98,5939	95,1826	75,3333	6,9703	80,1111	72,6667	83,7778	29,4403	83,314	71,4025	93,9209	53,2635

Cuadro 4: Datos del algoritmo GRASP

En la tabla 4 se encuentran los datos referentes a la ejecución del algoritmo GRASP sobre todas las bases de datos.

Vemos cómo este algoritmo consigue unas tasas de reducción bastante altas y, a la vez, unas clasificaciones en la función objetivo mejores que los algoritmos anteriores.

Esto se debe a que este algoritmo reúne las bondades del algoritmo voraz —alta reducción de características— y de la búsqueda local —mejores clasificaciones en la función objetivo—, consiguiendo así ya unos resultados más aceptables.

Además, este algoritmo es más rápido que BMB, así que su comportamiento —tanto en resultados como en eficiencia— es el más deseable de entre los vistos hasta ahora.

4.5. Búsqueda local reiterada

En la tabla 5 vemos los datos de la última metaheurística considerada: la búsqueda local reiterada.

Los resultados de este algoritmo deberían ser parecidos a los de la búsqueda multiarranque básica, ya que la idea detrás de su funcionamiento es parecida. A la vista de los resultados, es evidente que así es.

En este caso, sin embargo, se intenta añadir diversidad con la mutación de las soluciones. Esta diversidad es bastante positiva en lo referente a los tiempos: ILS tarda menos, en promedio, que BMB en encontrar las soluciones. Esto puede deberse a que las soluciones iniciales están más cerca

Particiones	WDBC				Movement Libras				Arrythmia			
	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	98,9437	96,1404	50	2,8364	75,5556	75	51,1111	10,4815	78,125	62,3711	54,6763	118,7117
Partición 1-2	97,8947	95,4225	36,6667	2,7133	76,6667	70	45,5556	11,2853	73,7113	64,0625	51,7986	87,9353
Partición 2-1	98,2394	96,1404	53,3333	2,7524	75,5556	74,4444	50	10,3327	73,4375	62,8866	53,2374	94,4518
Partición 2-2	98,5965	96,4789	50	2,6923	79,4444	79,4444	45,5556	13,7246	73,7113	65,625	51,0791	96,3821
Partición 3-1	98,5915	96,1404	53,3333	3,0409	79,4444	71,1111	52,2222	10,6698	77,0833	60,3093	45,6835	136,4909
Partición 3-2	97,8947	94,0141	56,6667	2,6608	81,1111	73,8889	42,2222	10,8124	73,7113	61,9792	52,1583	115,9396
Partición 4-1	98,9437	96,1404	46,6667	2,8189	73,8889	69,4444	56,6667	10,9984	76,0417	66,4948	51,0791	116,1311
Partición 4-2	98,2456	97,1831	33,3333	2,7649	76,6667	76,6667	50	10,7717	78,3505	62,5	51,4388	121,2225
Partición 5-1	97,8873	95,7895	56,6667	3,2199	75	73,3333	42,2222	12,3283	77,0833	63,9175	52,8777	104,7893
Partición 5-2	98,9474	94,0141	66,6667	2,4084	76,1111	76,6667	67,7778	11,5679	76,2887	66,6667	52,8777	114,7923
Medias	98,4185	95,7464	50,3333	2,7908	76,9444	74	50,3333	11,2973	75,7544	63,6813	51,6906	110,6847

Cuadro 5: Datos de la búsqueda local reiterada

de los óptimos locales, y la búsqueda local necesita de menos iteraciones para alcanzarla. ¿Por qué pasa esto? Al estar en zonas buenas del espacio de búsqueda, es probable que la mutación de la mejor solución encontrada en la iteración anterior se mueva a zonas mejores, haciendo que la búsqueda local converja más rápido.

Esta característica es muy positiva, ya que conseguimos resultados muy similares al BMB con una mejora en la eficiencia.

4.6. Datos generales

Particiones	WDBC				Movement Libras				Arrythmia			
	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
3-NN	96,2734	96,1681	0	$3{,}4 \cdot 10^{-3}$	68,8333	74,2222	0	$4{,}7 \cdot 10^{-3}$	63,423	62,7502	0	0,0189
SFS	97,2585	94,3029	84,3333	0,2043	76,2778	70,7222	90,1111	0,9234	78,5975	69,2886	97,3381	2,5154
BMB	98,348	95,9571	47,3333	3,2593	77,1111	74,5556	52,1111	12,2184	73,321	63,7274	50,7554	130,9448
GRASP	98,5939	95,1826	75,3333	6,9703	80,1111	72,6667	83,7778	29,4403	83,314	71,4025	93,9209	53,2635
ILS	98,4185	95,7464	50,3333	2,7908	76,9444	74	50,3333	11,2973	75,7544	63,6813	51,6906	110,6847

Cuadro 6: Datos generales

En la tabla 6 vemos un resumen de todos los datos obtenidos tras las ejecuciones de las metaheurísticas con las bases de datos.

Vemos ahora claro que el mejor algoritmo en lo referente a la reducción de características es, de lejos, el GRASP. Esta victoria es debida a su comportamiento voraz en la generación de soluciones iniciales.

Con respecto a la tasa de clasificación, los resultados son bastante variables con respecto a las bases de datos, y tomar una decisión sobre la bondad de uno u otro sería aventurarse demasiado: esta decisión debería ser tomada teniendo en cuenta la base de datos particular con la que se quiera trabajar. Aunque la tasa de clasificación dentro de la muestra de entrenamiento es claramente mejor en GRASP, parece que estamos cayendo en la trampa del sobreajuste: al mirar la tasa en la muestra de test, GRASP pierde en comparación con los otros dos algoritmos —excepto en Arrythmia, donde es el ganador absoluto en todos los aspectos—.

Por tanto, si tuviera que escoger uno y solo uno de los algoritmos considerados, elegiría GRASP: su tasa de reducción es la mejor de los tres y las tasas de clasificación son bastante aceptables. Esta decisión vendría

supeditada en casos reales, sin embargo, a un estudio pormenorizado del problema particular con el que estemos trabajando.